Ambiguity Reduction in Natural Language Requirements through Autoformalization

Eunsuk Kang, Sam Procter, Teresa Fortes, and Yining She

School of Computer Science, Carnegie Mellon University, Pittsburgh PA 15213
{eunsukk,sprocter,tfortes,yiningshe}@andrew.cmu.edu

Abstract. Natural language (NL) remains the primary medium for expressing system requirements, but such requirements are often ambiguous and admit multiple plausible interpretations. Recent work on LLM-based autoformalization translates NL requirements into formal specifications such as linear temporal logic, but involves choosing a particular interpretation, risking misalignment with developer intent. In this paper, we propose a different use of autoformalization: instead of producing one formal specification, we use the formal semantic domain to generate multiple candidate interpretations of an NL requirement and present them to developers as alternative NL interpretations. By allowing developers to select the interpretations that match their intent, the envisioned approach is intended to support iterative refinement of requirements and help reduce unintended ambiguity.

1 Introduction

The emergence and increasing power of LLMs bring new opportunities and needs for formal methods. In particular, researchers have recently been investigating the use of an LLM for *autoformalization* (e.g., [3, 1]); i.e., translating natural language (NL) requirements into formal specifications, such as linear temporal logic (LTL) [9]. Given that the difficulty in writing specifications is known to be a major limiting factor, this line of research has the potential to significantly improve the usability and adoption of formal methods.

However, there are two challenges to autoformalization of NL requirements. First, NL statements are inherently ambiguous and likely to admit multiple possible *interpretations*, one of which would need to be selected by the translation process; the problem then becomes validating whether the selected interpretation reflects what the user intended to mean, which is, on its own, a highly challenging task (known in the requirements engineering community as the *requirements validation* problem [2]). Second, although specifications have many downstream applications (such as verification and runtime monitoring), most human developers will still likely find it difficult to read and comprehend formal specifications [6] and prefer expressing, documenting, and debugging requirements in NL.

In this paper, we present another use of autoformalization for requirements engineering. Instead of generating formal specifications as the end product, we

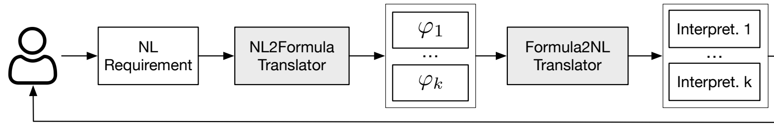

Fig. 1. An overview of ambiguity reduction approach through autoformalization.

advocate leveraging autoformalization as a way to help developers refine and improve their existing NL requirements through *ambiguity reduction*. An overview of the approach is shown in Figure 1. A developer begins by expressing a system requirement in NL; instead of producing a single logical formula (which corresponds to one particular interpretation of the requirement), the *NL2Formula* translator (implemented using an LLM) generates a set of k possible formulas (φ_1,...,φ_k), representing different ways in which the input requirement can be interpreted. These formulas, in turn, are translated back to their corresponding NL statements, to be presented to the developer as an **informal but precise** explanation of why their original requirement is ambiguous and how it could yield multiple interpretations. A key advantage of this back-translation is that it is grounded in the unambiguous semantics of the formal specification notation (e.g., LTL), making the resulting NL interpretations that are fed back to the user precise by construction (unlike the original requirement, which is subject to the inherent ambiguity of NL). The developer is then asked to select a subset (not necessarily just one) of these interpretations as their intended meaning, which are then merged into a revised version of the requirement that is now less ambiguous than the previous one.

In this process, the developer never directly sees the formal specifications; instead, the semantic domain of the specification notation is used as a formal basis to disambiguate and generate a space of possible interpretations of the original requirement. We also note that the goal of our approach is not to arrive at a requirement that is completely unambiguous; our viewpoint is that many system requirements will remain ambiguous to a certain degree (due to inherent subjectivity in the problem domain, under-specified or conflicting needs by users, etc.). Instead, the goal is to help the developer identify and refine parts of a requirement that are *unintentionally* ambiguous due to imprecision in NL.

2 Example

We illustrate the proposed approach and potential challenges that may be encountered using a small illustrative example. Consider the following hypothetical requirement in an avionics system: *If the pilot engages the manual brake, the system must disable the autopilot mode.*

Generating Candidate LTL Formulas (NL2Formula). The first stage of the proposed approach prompts an LLM to enumerate semantically distinct LTL interpretations of the given requirement. The prompt instructs the model to vary

three explicit *ambiguity dimensions*—timing, scope, and duration—and to use only a fixed set of LTL operators and atomic propositions (*engage*, *autopilot*). Restricting the operator set and proposition vocabulary serves two purposes: it reduces the search space for the model and bounds the risk of hallucinated formulas. As an experiment, we ran this prompt on three general-purpose LLMs—GPT-4o, Claude Sonnet 4.6, and Gemini 1.5 Pro—and recorded their outputs. GPT-4o and Claude Sonnet 4.6 converged on the same three semantically distinct formulas, varying timing (immediate vs. eventual) and duration (*engage until brake release*):

φ_1: $\square(engage \rightarrow \neg autopilot)$ *[timing: simultaneous]*
φ_2: $\square(engage \rightarrow \bigcirc\neg autopilot)$ *[timing: next-state]*
φ_3: $\square(engage \rightarrow (\neg autopilot\,\mathcal{U}\,\neg engage))$ *[duration: while engaged]*

Gemini, by contrast, replaced φ_3 with $\square(engage \rightarrow \lozenge\neg autopilot)$, missing the duration dimension entirely and producing a formula that is a weakened form of φ_2. This illustrates that general-purpose LLMs do not uniformly explore the ambiguity space and that prompt design (or specialized models) may be necessary to ensure adequate coverage of interpretation dimensions (see Section 4).

Back-translation to NL (Formula2NL). The three formulas are then translated back into NL by a second LLM prompt, which instructs the model to explain each formula without using the formal notation. Crucially, the logical structure and meaning of the back-translation is grounded in the semantics of LTL: each operator is translated to NL according to its definition ($\square$ as "always", $\bigcirc$ as "in the next state", and so on), while the atomic propositions are translated using the vocabulary terms drawn directly from the original requirement. This makes the back-translated interpretations *precise with respect to the formal semantics*, provided that the NL2Formula step produced a faithful formula. As an example, the result for φ_1 is:

I1 *"Whenever the pilot is engaging the manual brake, the autopilot must not be active at that same moment — the two can never be on simultaneously."*

Developer Selection and Feedback Loop. The developer is presented with the NL translations (I1–I3) and asked to identify which interpretations are consistent with their intent. In this example, the developer might indicate that none of the three captures their actual requirement and provide the following feedback: *"None of these capture that the autopilot must stay disabled continuously — not just be disabled at some point."* This feedback is passed back to NL2Formula, which then generates a revised set of interpretations. In our experiment, all three LLMs correctly identified $\square(engage \rightarrow \square\neg autopilot)$ as the primary refined interpretation — once the brake is engaged, autopilot must remain disabled permanently. GPT-4o and Claude Sonnet 4.6 additionally proposed:

φ_4 $\square(engage \rightarrow \bigcirc\square\neg autopilot)$ — permanent disabling beginning one step after engagement, capturing a plausible processing delay.

However, both models also produced a third formula with subtle issues: GPT-4o introduced $\Box(\mathit{engage} \rightarrow (\neg \mathit{autopilot}\ \mathcal{U}\ (\neg \mathit{engage} \wedge \Box \neg \mathit{autopilot})))$, which implicitly requires the brake to *eventually* be released—an assumption not present in the original requirement. Claude's third formula, $\Diamond(\mathit{engage}) \rightarrow \Box(\mathit{engage} \rightarrow \Box \neg \mathit{autopilot})$, is nearly equivalent to φ_4 in realistic scenarios where the brake is engaged at least once. This illustrates that models can introduce unintended assumptions or subtle redundancies during the feedback round, underscoring the need for equivalence checking as a post-processing step (see Section 4).

Through this iterative process, the developer was able to identify a missing assumption in the original requirement: it did not specify *when* autopilot could be re-enabled after being disabled. This motivates a new, more precise requirement: *Once disabled, autopilot mode must remain disabled until the pilot explicitly re-engages it.* This example demonstrates how the proposed approach can help not only disambiguate existing requirements but also reveal missing assumptions and drive the creation of new, more complete requirements.

3 Related Work

Structured Requirements Languages. NASA's Formal Requirements Elicitation Tool (FRET) [5] allows engineers to write requirements in FRETish, a structured natural language with precise, unambiguous semantics. For each FRETish requirement, FRET automatically generates temporal logic formulas, as well as natural language and diagrammatic explanations to help users validate their intent. While FRET shares our goal of bridging natural language and formal specifications, it requires requirements to be written upfront in a controlled language, whereas our approach accepts unconstrained NL as input and uses the formal semantic domain as an *internal* mechanism for disambiguation: the developer never directly authors or inspects formal specifications. However, FRETish could serve as a useful target representation for the alternative interpretations generated by our approach, providing a more structured and tool-supported medium for presenting candidates to developers.

Interactive Specification Synthesis. There are prior works on interactive synthesis of specifications. In particular, Gavran et al. [4] present a method for synthesizing LTL specifications from an NL description and a single example trace. Candidate specifications are generated and ranked, and *distinguishing traces* are shown to the user to resolve ambiguity through behavioral feedback. Our approach is complementary: rather than using traces as the medium for user interaction, we back-translate candidate formulas into NL explanations. Both approaches share the insight that a space of candidate interpretations must be explored interactively. The key difference lies in how that space is presented to the user.

LLM-based Autoformalization. Recent work has explored LLMs to translate NL requirements directly into temporal logic formulas [1, 3]. These approaches treat autoformalization as an end goal, producing a single formal specification. Our

work takes a different stance: we use autoformalization not as the end goal but as a mechanism for exposing latent ambiguities in NL requirements. By generating multiple semantically distinct candidate formulas and back-translating them into NL, we redirect the power of LLM-based formalization toward requirements refinement rather than specification generation.

4 Research Problems

Interpretation Generation. A key research challenge is how to carefully leverage the translation capabilities of an LLM. Generating too many alternatives may overwhelm the developer, while generating too few may fail to expose important sources of ambiguity. A naive use of an LLM may also produce formulas that are syntactically different but semantically equivalent. One promising direction is a neuro-symbolic approach where an LLM is used to propose candidate formulas, while formal analysis (e.g., logical equivalence checking) is used to filter redundant interpretations. This mirrors the *Generate-Verify-Repair* paradigm used in state-of-the-art approaches in LLM-assisted formal methods [8], and could be explored as an architectural pattern for the NL2Formula component.

Prompt Design and Model Specialization. As illustrated in Section 2, the quality and diversity of generated interpretations depends significantly on both the prompt quality and the underlying model. Our preliminary comparison of three LLMs shows that even with explicit guidance on ambiguity dimensions, models do not uniformly explore the interpretation space: Gemini 1.5 Pro omitted the duration dimension entirely on the simple avionics example, while GPT-4o and Claude Sonnet 4.6 converged on the same canonical set of formulas. For more complex requirements, possibly involving nested conditions, multiple temporal constraints, or probabilistic behaviors (e.g., PCTL [7]), general-purpose LLMs are likely to struggle further. A promising direction is the development of *specialized models* fine-tuned on requirement-formula pairs, which could provide more reliable coverage of the interpretation space while reducing hallucination risks.

Supporting Interactive Refinement. There are different ways in which generated interpretations can be presented to the developer. One possible approach is to display literal translations of the LTL formulas, as described in Section 2. Another approach could instead highlight the parts of the original requirement that are ambiguous; e.g., *It is ambiguous whether the autopilot mode must be (1) disabled immediately in the next system state or (2) disabled eventually at some time in the future.* Yet a third approach may involve showing sample system traces that distinguish a pair of interpretations. Evaluating the effectiveness of these different interaction methods through a user study would be valuable.

Beyond the initial presentation of interpretations, a key design question is how to support the developer when *none* of the generated candidates matches their intent. One approach may involve an iterative *feedback loop* in which the developer provides natural language feedback describing what is missing or incorrect, which is then incorporated into a revised prompt for the NL2Formula

component. The feedback loop also revealed a secondary ambiguity: the question of *when* autopilot can be re-engaged, which is not addressed by any of the generated formulas and motivates a new requirement. An open research question is how to structure feedback prompts to maximize the likelihood of convergence while minimizing the number of interaction rounds required.

Supporting Different Specification Styles. While the example in this paper uses LTL as the target specification notation, the proposed approach is not limited to a particular specification notation. Specifying requirements in different application domains (or different aspects of a single system) may involve different styles of specifications, such as temporal logic, contracts with pre- and post-conditions, state machines, or scenario-based models. Each of these notations defines a semantic space that can serve as a basis for generating alternative interpretations of an NL requirement. Beyond behavioral specifications, the similar idea could also apply to non-functional requirements (e.g., performance or security) expressed in architectural or policy-oriented languages. Integrating the proposed approach into an AI-based development framework, to help the developer iteratively refine and improve their prompts (which embody requirements in different styles), is also another interesting research direction.

References

1. Chen, Y., Gandhi, R., Zhang, Y., Fan, C.: NL2TL: Transforming natural languages to temporal logics using large language models. In: ENMLP. pp. 15880–15903. Association for Computational Linguistics (Dec 2023)
2. Cheng, B.H.C., Atlee, J.M.: Research directions in requirements engineering. In: 2007 Future of Software Engineering. p. 285–303. IEEE Computer Society (2007)
3. Cosler, M., Hahn, C., Mendoza, D., Schmitt, F., Trippel, C.: nl2spec: Interactively translating unstructured natural language to temporal logics with large language models. In: Computer Aided Verification. pp. 383–396 (2023)
4. Gavran, I., Darulova, E., Majumdar, R.: Interactive synthesis of temporal specifications from examples and natural language. Proc. ACM Program. Lang. **4**(OOPSLA) (Nov 2020). https://doi.org/10.1145/3428269, `https://doi.org/10.1145/3428269`
5. Giannakopoulou, D., Pressburger, T., Mavridou, A., Rhein, J., Schumann, J., Shi, N.: Formal requirements elicitation with FRET. In: REFSQ Workshops (2020), `https://api.semanticscholar.org/CorpusID:214708107`
6. Greenman, B., Saarinen, S., Nelson, T., Krishnamurthi, S.: Little tricky logic: Misconceptions in the understanding of LTL. The Art, Science, and Engineering of Programming **7**(2) (Oct 2022)
7. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Form. Asp. Comput. **6**(5), 512–535 (Sep 1994). https://doi.org/10.1007/BF01211866, `https://doi.org/10.1007/BF01211866`
8. Narodytska, N.: Integrating large language models in automated program verification. In: 2025 Formal Methods in Computer-Aided Design (FMCAD). pp. 4–4 (Oct 2025). https://doi.org/10.34727/2025/isbn.978-3-85448-084-6˙4
9. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science. pp. 46–57 (1977)